Health Plan Management Configuration System Design Document

1. Executive Summary

This document outlines the design for a configuration management system within a Health Plan Management application. The system handles hierarchical and inheritable configurations across entities such as Tenants, Carriers, Market Places, Market Segments, and Product/Module settings. It supports base templates (core product templates) and carrier-specific templates, enabling default settings at each level with inheritance to reduce redundancy and ensure consistency.

Key features include:

- **Hierarchical Inheritance**: Configurations cascade from higher levels (e.g., Carrier) to lower ones (e.g., Market Place, Market Segment), with overrides possible at child levels.
- **Template Management**: Reusable base templates for core products and customizable carrier-specific ones.
- **Data Model**: A hybrid approach using PostgreSQL's relational model for structured entity relationships and JSONB for flexible, document-like configuration storage.
- **Architecture**: Microservices backend in Java Spring Boot, React-based frontend for configuration management, and a Java SDK for runtime access to configurations, optimizing performance by avoiding direct API calls during application execution.
- **Performance Optimization**: SDK enables efficient config retrieval (e.g., via caching or in-memory access) for runtime operations.

The design ensures scalability, maintainability, and ease of extension for health plan configurations, such as those for tenants like Cambria operating in multiple market places and segments.

2. Design Overview

2.1 Problem Statement

Health plan management involves complex configurations that vary by tenant/carrier, market place, market segment, and product/module. Configurations must be inheritable to avoid

duplication, with support for templates to standardize core settings while allowing customizations. The system needs:

- A user-friendly interface (React frontend) for admins to manage configs.
- Efficient backend services (Java Spring microservices) for CRUD operations.
- High-performance runtime access (via SDK) to prevent bottlenecks in application flows.
- A robust data model balancing structure and flexibility.

2.2 High-Level Architecture

- **Frontend**: React application for viewing, editing, and managing configurations. It interacts with backend APIs via RESTful calls.
- **Backend**: Microservices built with Java Spring Boot, handling authentication, validation, inheritance resolution, and persistence.
- **Data Storage**: PostgreSQL database with relational tables for entities and JSONB columns for dynamic configs.
- **Runtime SDK**: A Java library that applications can integrate to fetch resolved configurations efficiently (e.g., using caching mechanisms like Redis or in-memory stores).
- **Integration Points**:
    - Frontend → APIs (REST/HTTP) for management.
    - Runtime Applications → SDK (method calls) for config access, which internally queries services or cache.

The system follows a microservices pattern for modularity:

- Config Management Service: Handles CRUD and inheritance logic.
- Template Service: Manages base and custom templates.
- Runtime Service: Exposes configs for SDK, with caching.

2.3 Key Principles

- **Inheritance**: Parent configs (e.g., Carrier) provide defaults; children (e.g., Market Place) can override or extend.
- **Templates**: Base templates define core settings; carrier-specific ones inherit and customize.
- **Idempotency and Validation**: Ensure configs are consistent and valid across levels.
- **Security**: Role-based access control (RBAC) for tenants/carriers.
- **Performance**: SDK uses pre-resolved or cached configs to minimize latency.

3. Configuration Structure and Examples

3.1 Config Hierarchy

The configuration is structured as a tree:

- **Root: Tenant/Carrier** (e.g., Cambria) – Global defaults.
- **Level 1: Market Place** (e.g., New Jersey, Georgia) – Inherits from Carrier, overrides as needed.
- **Level 2: Market Segment** (e.g., Small Group, ICHRA) – Inherits from Market Place.
- **Level 3: Product/Module** (e.g., Medical, Ancillary; modules like group profile, census) – Inherits from Market Segment, with specific settings.

Each level has:

- **Default Settings**: Key-value pairs or structured objects (e.g., eligibility rules, pricing params).
- **Overrides**: Explicit changes at lower levels.
- **Templates**: Applied at creation or update.

3.2 Config Format

Configs are stored as JSON objects for flexibility. Example structure:

```
{
  "tenantId": "cambria",
  "carrierSettings": {
    "defaultEligibility": {"ageMin": 18, "ageMax": 65},
```

```
   "billingCycle": "monthly"
  },
  "marketPlaces": [
   {
    "id": "new-jersey",
    "overrides": {"billingCycle": "quarterly"},
    "marketSegments": [
     {
      "id": "small-group",
      "overrides": {"groupSizeMin": 2, "groupSizeMax": 50},
      "products": [
       {
        "id": "medical",
        "modules": {
         "groupProfile": {"enabled": true, "fields": ["name", "address"]},
         "census": {"enabled": true, "validationRules": ["ssnRequired"]}
        }
       },
       {
        "id": "ancillary",
        "modules": {...}
       }
      ]
     }
    ]
   }
  ]
}
```

3.3 Inheritance Resolution

- When fetching a config for a leaf node (e.g., Medical product in Small Group, New Jersey), merge parent configs bottom-up.

- Algorithm: Start from leaf, apply overrides, then inherit unset fields from parent, recursively.

Example:

- Carrier default: {“billingCycle”: “monthly”}

- Market Place (NJ) override: {“billingCycle”: “quarterly”}

- Resolved for NJ Small Group Medical: {“billingCycle”: “quarterly”} (overrides carrier).

3.4 Templates

- **Base Template**: Core product settings, e.g., for Medical: {“modules”: {“groupProfile”: {“enabled”: true}}}
- **Carrier-Specific**: Inherits base, adds customizations, e.g., Cambria adds {“eligibility”: {“customRule”: “stateResident”}}
- Application: When creating a new Market Place config, apply template and allow edits.

4. Data Model

Using PostgreSQL for a hybrid relational-document model:

- Relational tables for entity relationships and fixed attributes.
- JSONB columns for dynamic configs to handle varying structures without schema changes.

4.1 Relational Tables

1. **Tenants** (or Carriers, assuming 1:1 mapping):
    - id (UUID, PK)
    - name (VARCHAR, e.g., “Cambria”)
    - config_template_id (FK to Templates)
    - created_at, updated_at (TIMESTAMPS)
2. **MarketPlaces**:
    - id (UUID, PK)
    - tenant_id (FK)
    - name (VARCHAR, e.g., “New Jersey”)
    - parent_config_id (FK for inheritance, optional)
    - config (JSONB: overrides and defaults)

3. **MarketSegments**:
    - id (UUID, PK)
    - market_place_id (FK)
    - name (VARCHAR, e.g., “Small Group”)
    - config (JSONB)
4. **Products**:
    - id (UUID, PK)
    - market_segment_id (FK)
    - type (VARCHAR, e.g., “Medical”)
    - config (JSONB: module settings)
5. **Templates**:
    - id (UUID, PK)
    - type (ENUM: ‘base’, ‘carrier-specific’)
    - name (VARCHAR)
    - config (JSONB: template data)
    - applies_to_level (ENUM: ‘carrier’, ‘marketplace’, etc.)

4.2 Indexes and Constraints

- Indexes on FKs and JSONB fields (e.g., GIN index on config for fast queries: CREATE INDEX idx_config ON market_places USING GIN (config);)
- Unique constraints: e.g., unique (tenant_id, name) for MarketPlaces.
- Triggers: For inheritance validation or auto-merging on save.

4.3 Query Examples

Fetch resolved config for a product: Use recursive CTE to merge JSONB from hierarchy.

```
WITH RECURSIVE config_hierarchy AS (
```

```
  SELECT config FROM products WHERE id = 'product_id'
  UNION ALL
  SELECT ms.config FROM config_hierarchy ch
  JOIN market_segments ms ON ch.market_segment_id = ms.id
  -- Continue up to carrier
)
SELECT jsonb_merge_deep(...) FROM config_hierarchy;  -- Custom function for deep merge
```

-

5. Architecture Details

5.1 Components

- **Frontend (React)**: Admin dashboard with forms for editing hierarchies, templates. Uses libraries like React Hook Form, Material-UI. API calls via Axios.
- **Backend Microservices (Java Spring Boot)**:
    - **Config Service**: REST APIs for CRUD (/api/configs/{level}/{id}). Handles inheritance resolution in business logic.
    - **Template Service**: APIs for template management (/api/templates).
    - **Auth Service**: JWT-based authentication.
    - Dependencies: Spring Data JPA for relational, Spring Data JDBC for JSONB, Lombok for boilerplate.
- **SDK (Java Library)**:
    - Methods like Config getResolvedConfig(String tenantId, String marketPlaceId, …);
    - Internally: Calls Config Service or uses Redis cache for pre-resolved configs.
    - Avoids HTTP overhead; uses gRPC if inter-service, or direct method calls if monolith.
- **Database**: PostgreSQL cluster for HA.
- **Caching**: Redis for resolved configs, invalidated on updates.
- **Deployment**: Docker/Kubernetes for microservices.

5.2 Communication

- Frontend ↔ Backend: REST over HTTPS.
- Runtime Apps ↔ SDK: Library integration (e.g., Maven dependency).
- Inter-Service: gRPC for performance if needed.

6. Design Sequence Flows

6.1 Sequence Diagram: Creating a New Configuration (Admin Flow)

1. Admin logs in via React UI.
2. UI → API: GET /api/templates?type=base (fetch available templates).
3. Config Service → DB: Query Templates table.
4. UI displays form; Admin selects template, edits overrides.
5. UI → API: POST /api/marketplaces (with tenantId, name, config JSON).
6. Config Service: Validate, apply template, merge inheritance, persist to DB.
7. Service → DB: INSERT into MarketPlaces.
8. Response: Success, updated config.

6.2 Sequence Diagram: Runtime Config Access

1. Application Runtime initializes SDK.
2. Runtime → SDK: getResolvedConfig(tenantId="cambria", marketPlace="new-jersey", segment="small-group", product="medical").
3. SDK checks cache (Redis): If hit, return.
4. If miss: SDK → Config Service (internal call/gRPC): Resolve config.
5. Service → DB: Recursive query to merge hierarchy.
6. Service caches result in Redis.
7. SDK returns resolved JSON to Runtime.

6.3 Sequence Diagram: Updating a Config with Inheritance

1. UI → API: GET /api/configs/marketplace/{id} (fetch current).
2. Service → DB: Fetch with parents.
3. UI edits overrides.
4. UI → API: PUT /api/configs/marketplace/{id} (new config).
5. Service: Merge with parent, validate no conflicts, update DB.
6. Invalidate cache for affected hierarchy.

6.4 Error Handling Flow

- Validation errors (e.g., invalid override): Return 400 Bad Request.
- Inheritance conflicts: Log and notify admin.
- DB failures: Retry with Spring Retry.

7. Implementation Recommendations

- **Phased Rollout**: Phase 1: Core data model and APIs. Phase 2: Inheritance logic. Phase 3: SDK and caching. Phase 4: React UI integration.
- **Testing**: Unit (JUnit), Integration (Postman), E2E (Cypress for UI).
- **Monitoring**: Prometheus + Grafana for metrics, ELK for logs.
- **Scalability**: Horizontal scaling of services; sharding DB by tenant if needed.
- **Future Extensions**: Add versioning for configs, audit logs.

This design provides a solid foundation for managing complex health plan configurations efficiently and scalably.

Addendum: Detailed Sequence Diagrams for Health Plan Management Configuration System

This addendum expands on Section 6 of the original design document, providing more granular sequence diagrams for key flows. Sequence diagrams are described using a textual notation for clarity, following UML-like conventions. Participants include:

- **Admin User**: Interacts via React Frontend.
- **React Frontend**: UI layer handling user inputs and API calls.
- **Config Service**: Backend microservice for configuration management.
- **Template Service**: Backend microservice for template operations (if separate; otherwise, integrated into Config Service).
- **Auth Service**: Handles authentication (assumed integrated or separate).
- **Database (PostgreSQL)**: Persistence layer.
- **Cache (Redis)**: For performance optimization.
- **SDK**: Runtime library used by application components.
- **Runtime Application**: The health plan management app consuming configs.

Each diagram includes activation bars, synchronous calls (→), asynchronous if applicable (⇢), returns (←), and notes for explanations. Error paths are included where relevant.

6.1 Detailed Sequence Diagram: Creating a New Configuration (Admin Flow)

This flow covers an admin creating a new Market Place configuration under a Tenant/Carrier, applying a template, and persisting with inheritance.

Admin User -> React Frontend: Login and navigate to Config Management Dashboard

React Frontend -> Auth Service: POST /auth/login (username, password)

Auth Service -> Database: Validate credentials (SELECT from users)

Auth Service <-- Database: User details

Auth Service -> React Frontend: JWT Token (200 OK)

Note: Session established with JWT for subsequent calls.

Admin User -> React Frontend: Select "Create Market Place" for Tenant "Cambria"

React Frontend -> Template Service: GET /api/templates?type=carrier-specific&tenantId=cambria (Authorization: Bearer JWT)

Template Service -> Database: Query Templates table (SELECT WHERE type='carrier-specific' AND applies_to_level='marketplace')

Template Service <-- Database: List of templates (e.g., JSONB configs)

Template Service -> React Frontend: 200 OK with templates array

React Frontend -> Admin User: Render form with template dropdown and editable fields (e.g., name="New Jersey", overrides JSON editor)

Admin User -> React Frontend: Fill form, select template, edit overrides (e.g., {"billingCycle": "quarterly"}), submit

React Frontend -> Config Service: POST /api/marketplaces (body: {tenantId: "cambria", name: "New Jersey", templateId: "template-123", overrides: {...}}, Authorization: Bearer JWT)

Config Service -> Auth Service: Validate JWT (internal call or filter)

Config Service -> Template Service: GET /api/templates/{templateId} (fetch template config)

Template Service -> Database: SELECT config FROM templates WHERE id='template-123'

Template Service <-- Database: Template JSONB

Template Service -> Config Service: Template data

Config Service: Business Logic - Merge template with overrides, resolve inheritance from parent (Carrier config)

Config Service -> Database: Fetch parent config (SELECT config FROM tenants WHERE id='cambria')

Config Service <-- Database: Parent JSONB

Config Service: Deep merge (parent + template + overrides), validate (e.g., no invalid keys)

alt Validation Fails

Config Service -> React Frontend: 400 Bad Request (error: "Invalid override key")

else Validation Succeeds

Config Service -> Database: INSERT INTO marketplaces (id=UUID(), tenant_id='cambria', name='New Jersey', config=merged JSONB)

Database --> Config Service: Insert success

Config Service -> Cache: Invalidate/Purge keys for hierarchy (e.g., DEL redis_key:"config:cambria:*")

Config Service -> React Frontend: 201 Created (with new marketplace ID and full resolved config)

end

React Frontend -> Admin User: Display success message and updated config view

Notes:

- Total steps: ~15 interactions for a successful creation.
- Time estimates: UI rendering <1s, API calls ~200-500ms each, DB operations ~10-50ms.
- Security: All API calls include JWT; Config Service checks tenant ownership.

6.2 Detailed Sequence Diagram: Runtime Config Access

This flow shows how the runtime application fetches a resolved config for a specific product via SDK, with caching.

Runtime Application -> SDK: Initialize SDK (e.g., ConfigSDK.init(baseUrl, apiKey))

Note: SDK configured with service endpoints and cache client.

Runtime Application -> SDK: Config resolvedConfig = sdk.getResolvedConfig(tenantId="cambria", marketPlaceId="new-jersey", segmentId="small-group", productId="medical")

SDK -> Cache: GET redis_key:"resolved_config:cambria:new-jersey:small-group:medical"

alt Cache Hit

  Cache -> SDK: Cached JSON

  SDK -> Runtime Application: Return Config object (parsed JSON)

else Cache Miss

  SDK -> Config Service: Internal call (gRPC or HTTP) /internal/resolve-config (params: tenantId, marketPlaceId, etc.)

  Config Service -> Auth Service: Validate API key or internal auth

  Config Service -> Database: Recursive CTE query to fetch hierarchy (WITH RECURSIVE ... SELECT jsonb_merge_deep(...))

  Database -> Config Service: Raw hierarchy data (JSONB arrays)

  Config Service: Business Logic - Bottom-up merge (product + segment + marketplace + carrier + base template if needed)

  Config Service: Validate resolved config completeness

  Config Service -> SDK: Resolved JSON (200 OK)

  SDK -> Cache: SET redis_key with TTL (e.g., 1 hour) and value=resolved JSON

  SDK -> Runtime Application: Return Config object

end

Notes:

- Caching reduces latency: Cache hit ~1ms, miss ~100-300ms.
- SDK handles serialization (e.g., to POJO classes like MarketPlaceConfig).
- Fallback: If service down, SDK could use local fallback or last-known config.

6.3 Detailed Sequence Diagram: Updating a Config with Inheritance

This flow details updating a Market Segment config, propagating inheritance checks.

Admin User -> React Frontend: Navigate to Edit Market Segment "small-group" under Market Place "new-jersey"

React Frontend -> Config Service: GET /api/marketsegments/{segmentId} (Authorization: Bearer JWT)

Config Service -> Database: Fetch segment config (SELECT * FROM market_segments WHERE id={segmentId})

Config Service <-- Database: Segment data

Config Service: Resolve full inheritance (internal merge with parents)

Config Service -> React Frontend: 200 OK with resolved config and overrides

React Frontend -> Admin User: Render editable form showing current overrides and inherited values (grayed out)

Admin User -> React Frontend: Edit overrides (e.g., add {"groupSizeMax": 100}), submit

React Frontend -> Config Service: PUT /api/marketsegments/{segmentId} (body: {overrides: new JSON}, Authorization: Bearer JWT)

Config Service -> Auth Service: Validate JWT

Config Service -> Database: Fetch current config and parents (multi-query or join)

Config Service <-- Database: Hierarchy data

Config Service: Merge new overrides with existing, re-resolve against parents, validate (e.g., no override of immutable fields)

alt Validation Fails

  Config Service -> React Frontend: 400 Bad Request (error details)

else Validation Succeeds

Config Service -> Database: UPDATE market_segments SET config=newOverridesJSONB WHERE id={segmentId}

Database --> Config Service: Update success

Config Service -> Cache: Invalidate affected keys (e.g., DEL "resolved_config:cambria:new-jersey:small-group:*")

Config Service: Optional - Propagate notifications (e.g., async event to audit log or dependent services)

Config Service -> React Frontend: 200 OK with updated resolved config

end

React Frontend -> Admin User: Refresh view with success

Notes:

- Inheritance ensures updates don't break children; if children exist, warn or auto-propagate overrides.
- Versioning: Add optimistic locking (e.g., etag or version column) to prevent concurrent updates.

6.4 Detailed Sequence Diagram: Error Handling Flow

This is a cross-cutting flow, shown for a generic API error during config creation/update.

Actor (e.g., React Frontend or SDK) -> Config Service: API Call (e.g., POST /api/marketplaces with invalid data)

Config Service: Initial validation (e.g., schema check with Jackson/JSON Schema)

alt Input Validation Error

Config Service -> Actor: 400 Bad Request (JSON error: {field: "name", message: "Required"})

else Proceed

Config Service -> Database: Attempt operation (e.g., INSERT)

alt DB Constraint Violation (e.g., unique name)

```
    Database -> Config Service: SQLException (e.g., duplicate key)
    Config Service: Log error (SLF4J to ELK)
    Config Service -> Actor: 409 Conflict (error: "Name already exists")
  else Business Logic Error (e.g., inheritance conflict)
    Config Service: Detect in merge logic
    Config Service -> Actor: 422 Unprocessable Entity (error: "Override conflicts with parent")
  else External Service Error (e.g., Template Service down)
    Config Service -> Template Service: GET (times out)
    Config Service: Retry (Spring Retry: 3 attempts, exponential backoff)
    alt Retry Fails
      Config Service: Log and circuit break (if Resilience4j used)
      Config Service -> Actor: 503 Service Unavailable (retry-after header)
    end
  end
end

Note: Global exception handler in Spring catches unhandled errors -> 500 Internal Server Error
Config Service -> Monitoring: Send metrics (Prometheus: error counter increment)
```

Notes:

- Use standardized error responses (RFC 7807 Problem Details).
- Client-side: React handles errors with toast notifications, retry buttons.
- Recovery: For transient errors, implement idempotency (e.g., request IDs).